CS246: Watopoly Design Document

Members: Lizzie Kim, Emily Shao, David Lu

August 15, 2020

Introduction

The CS246 group involving members Lizzie Kim (j575kim), David Lu (j333lu), and Emily Shao (ye2shao) decided to take on the project Watopoly which is a mock version of Monopoly, a real board game that is played by many people today.

This is a board game with different squares that players can land on. Each cell represents some kind of property that players can buy, trade, and sell. It is a game where players can roll dice, move around the board, and try to earn the most money and assets as they can. Ultimately, the final person standing who hasn't gone bankrupt is the winner.

Overview (overall structure of the project)

Watopoly was written in C++ with Object Oriented Design principles.
The structure of the project revolves around three important design patterns: inheritance, observers, and polymorphism. The group was able to utilize the fact that many of the characteristics of the game were similar or repeated.

1. The game is controlled through the command line. As an interactive game, it receives inputs from the user to access player movement as well as file handling.
2. The official game is run in the main file. When the game is booted up, it will create an instance of the game board and the players who are playing. However, any changes to the game board and the players participating in the game are kept track in the board object as well as the player objects.
3. The gameboard is the subject that watches the player movement, while the cells are the observers waiting to hear about this change. Whenever a player moves their position, it alerts the cells, and as the cells know that it is time to check whether the player has landed on them, they check to see if the player's position is the same as theirs. If it is the same, they carry out their respective actions to the player.
4. The cells are divided into different classes as they have different characteristics. The cell is mainly divided into property and non-property, where the property is a building you can buy, and non-property is a space that is not bought but still has an action to carry out. The properties divide further into academic buildings, gyms, and residences. They are different since buildings can be improved, while gyms and residences cannot.
5. The player class also has a vector of cell objects in order to keep track of the properties they own.

Design (Specific techniques)

There were a few techniques the group emphasized on using to solve various design challenges:

1. **Challenge:**
   We noticed that the individual cells on the board shared many similarities. Thus, we had to find the most efficient way to add all 40 cells as subclasses to the parent Cell class.

   Solution: Take advantage of inheritance relationship features
   Inheritance was a huge part of the design. Inheritance is important due to the reusability. Inheritance makes sense because we are recognizing that certain objects have similar behaviour and have a relationship. As we implement one parent class, we are able to easily implement some child classes where they are able to take after their parent and have a similar form, but at the same time, have some flexibility to change for their specific needs. In our case, we were able to recognize that the cells on the board all had similar behaviour. Within cell, they branched into more classes which could be classified as Property and Non-property. Although the two are similar in the sense that they are both places that the user can land on and interact with, they were different in the way that Property can be bought while Non-property cannot be. Furthermore, these two classes were able to appropriately branch off into more classes such as Academic, Gym, and Residences for the Property class. We were able to reuse code and have a system that made sense. The code was able to be cohesive as the classes had well-defined relationships for systematic clarity.

2. **Challenge:**
   We knew that it wasn't efficient or clean practice to rewrite the same lines of code over and over again, so we had to find a way to prevent this.

   Solution: Create utility functions within classes
   Thinking about reusability was a huge part of the group. Therefore, we tried to create utility functions within classes whenever appropriate. For example, in our board, we created a function called findCell. Whenever a player moved or wanted to interact with a certain part of the board, we needed to retrieve that appropriate cell. To do so, we were iterating through the cells of the board until we found the one that was being requested. As we made this into a function that could be easily called, there was less repetitive code and better reusability.

3. **Challenge:**
   We wanted to be able to find a way to notify our system of changes that occurred as players entered their commands.

   Solution: Integrate observer pattern into our code structure
   The group was able to effectively take advantage of the observer pattern. The observer pattern is best practice for when we want to create a one-to-many

dependency between objects (Learn Module). A subject is essentially a class where it is watching for some sort of behaviour that we want to keep track of. If there is a change in that specific thing that we were looking out for, the classes that must know of this change are notified. The group was able to use this for player movement and cells on the board. Whenever a player would roll the dice and move, then they would land on a new cell on the board. The entire purpose of the game is for a player to interact with the cells they are landing on and choose certain actions to win. Therefore, whenever a player would land on a cell, we would notify that cell that a player has landed and then the cell would perform the right actions. Essentially, our board was the subject that kept track of where the player was going, and our cell was the observer where it would be notified of the player movement. Through the observer pattern, we were able to create better cohesion and less coupling. Instead of having to create extra objects, functions, or classes, we were able to give the existing classes a relationship that was meaningful and made sense.

4. **Challenge:**
   Because of our heavy usage of inheritance, we were sometimes unable to access certain methods that we needed. We wanted to re-use the same functions as much as possible.

   Solution: Use dynamic casting and polymorphism
   The group was able to make great usage of polymorphism and dynamic casting. As the group made a huge use of inheritance, there were different parent and child objects that we had to deal with. In order to get around with using functions, we were able to use casting to access functions as we needed. For instance, we were able to keep a polymorphic array of Cells. This means the array can have objects of Cells, but that means they can be in the form of Properties, non-properties, etc. However, whenever we would access a cell and we knew that it would be a property, how could we perform the property actions? We were able to use casting for that usage.

Changes to Original Design

1. **Observers**
   We currently have the board as the observer, notifying the cells (subjects) about player movement. Our original design plan was to have a display class, with the class being the subject, while the cells and board would notify it of any changes, and the display would then change accordingly. However, we never ended up implementing a display class, so we changed our observer plan.

2. **Functions / Variables**
   There were many functions that we never thought to use, or ones which we thought wouldn't be needed, that we ended up implementing into our final project. These functions were mostly for accessing certain elements within classes, such as finding properties and owners of properties, which made our lives a lot easier. Of course there were also functions and variables which ended up being redundant, so those

ended up either being removed or replaced. An example was that originally we were planning to have a posX and a posY instance variable in our Cell class, but we realized that it would be easier to just use an integer as our position, since our board would be the same every time, making it easier to track.

Resilience to Change (how it supports possibility to change of program specification)

1. **Vectors**

   One important feature the group was able to utilize was vectors. By using vectors, the group was able to flexibly add to it and remove from it whenever necessary. Therefore, if specifications changed where there were more cells needed or players, then it could be done quite easily. For instance, for one of the functions, we used an array of bool. This means that the array was initialized with a hard coded number. This is where we saw the importance of vectors -- we could not flexibly change the array as we needed it to, and therefore, we had to use vectors in order to take advantage of resizing. This helped us with easy access and easy changes as we needed.

2. **Inheritance**

   In addition, the group made a great usage of inheritance. By using this object oriented programming principle, the group can make copies of certain buildings. For example, if Waterloo built another building, then we would be able to find the appropriate umbrella for it to go under. As we can reuse the code and help them find a relationship with each other as appropriate, it is easy to create the objects.

3. **Design Pattern**

   Lastly, like stated before, the group used the observer pattern. This means that even if we add another cell or another type of building, we can still interact with the cell easily. If we had hardcoded checking each position and performing the right actions, it would take us a long time if the specifications changed. However, as we let the board notify the cells wherever a player has landed, even if we add more cells or different types of cells, the functionality still remains the same.

Project-Specific Questions

1. **Question. After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a gameboard? Why or why not?**

   Previously, for DD1, we had written that the Observer Pattern would be a great design pattern for board visualization. Although we still believe that the Observer Pattern is a good pattern to use for when implementing a gameboard, we thought that it was useful for our player movement and cell interaction. The observer pattern is good to use when you want to have a one to many dependency. This means that if there is some sort of change that many classes have to look for and it is the same change, it is an efficient design pattern to use. We realized that each cell is also an object that is looking to carry out some action. For instance, the goose nesting must tell the user

they have been attacked by geese. On the other hand, a property may want to tell the user that they need to pay up. In the end, they all need to look out for player movement to see if the player position is equivalent to their own, and if they are, then to carry out these actions. Therefore, we believed the Observer Pattern is a good pattern to use for implementing cells.

2. Suppose that we wanted to model SLC and Needles Hall more closely to Chance and Community Chest cards. Is there a suitable design pattern you could use? How would you use it?

If we wanted to model SLC and Needles Hall more closely to Chance and Community cards, we would still use the factory pattern method. SLC and Needles Hall are both based on a probability, giving the user some random outcome. If we were to model them like cards, then the factory pattern method is perfect since we just create a card every time we need them. They aren't objects that are created and will remain, but are objects that come and go. Therefore, this design pattern is perfect since we have reusability with the similarity they have in random-ness, but also in the sense that we get them when we need them.

3. Question. Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?

After working on this project, it does seem like that the decorator pattern would be a good pattern to use when implementing improvements. Previously, prior to DD2, the group had thought that improvement wasn't an important characteristic at all, and wouldn't have much effect on the way objects behave. Therefore, the group thought that it would not be a good idea to implement improvements with the decorator pattern since it was unnecessary work. However, one problem that the group came across was having high coupling since there was a lot of inheritance going on to accommodate for the differences some objects had, including Improvements. This means that we had to create a lot of subclasses (i.e) create classes for MKV, V1, REV, etc). It felt like we had a lot of "meaningless" classes because they weren't doing anything special in particular. Therefore, if we used the decorator pattern, we could give objects the ability to add on functionality without being a whole separate class. Improvement is a good example of this since it is one additional characteristic of a cell, as some cells are able to be improved while some cannot (like academics can be improved but SLC can't.). By making usage of composition, we could have added this functionality as necessary.

Extra features:

1. Additional Commands (For better gameplay)

The group was able to implement some helpful commands such as the "help" function. Through help, the user can easily see all the commands they are able to use, what each commands do, and how to use them. However, this only pops up if the user asks for help, which means that we aren't crowding the screen with all these commands each turn. Not only this, but the group took some user experience into consideration. For instance, when a user lands on a property, we are able to provide some helpful information such as the name of the property. Even if we didn't have this, the game would still be able to carry out the necessary functions with the user's inputs. However, in order to make the user's gameplay as smooth as possible, we were able to put these little bits of information.

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

There were three challenges that the group found while developing software in teams. First, the group realized that communication is significant. Especially with our course being online and most people working from home, we had to work on this project virtually. That means that communication could not be as efficient. However, we tried our best to give daily updates on what we have worked on, what we are stuck on, and what needs to be completed. By doing so, we did not have overlapping work, and helped unblock each other by looking over each other's code and debugging together. Second, version control was very important in keeping our clean orderly. The members would be working on different tasks in parallel, but our work was important to piece together appropriately. Therefore, if we would pull, we would make sure to rebase, and before pushing or making merge requests, we would be sure to solve conflicts. We wanted to avoid cases where code would be lost, or someone would write over existing functions. These lessons were essential in helping us learn technical soft skills. Building software in teams is not an easy task -- one thing needs to be done in order for the next thing to run. If there is no proper communication within the team, work becomes less productive, and more confusing. Each person has their own strength to play, and if you are able to utilize that appropriately by effective communication, then the output will be more outstanding than done alone.
Lastly, what this project taught us was the importance of having proper documentation of your code. When programming by yourself, you can get away with having bad/no documentation most of the time, as you're the only one working on it, and you'll be able to keep track of what you've been doing for the most part. However,

working in a group stresses the importance of proper documentation, so other team members have an idea of what your code is trying to do, and will be able to add code on top of it, or utilize your code without confusion.

2. **What would you have done differently if you had the chance to start over?**

   If we had the chance to start over, we would definitely design the program step by step while mocking the gameplay. As reflected in the modified answers to project specific questions, the group began to have different ideas and approaches as we understood the project more deeply. Although the group was able to thoroughly read through the instructions and understand the rules, there were many rules we did not discover until after they were implemented the wrong way. Therefore, we came across challenges while debugging where we found that certain utility functions were needed or we were missing certain fields. This means that we did not have the chance to maximize low coupling and high cohesion as much as we potentially could have, and had to improvise some functions along the way whenever we realized we needed them. Our group did a good job overall of naming functions and variables, however we could've done a better job of commenting and documenting our code.

Conclusion

In conclusion, the project Watopoly was a good way to test our software developing skills through making decisions on good design patterns, rewriting clean code, and testing.

Importantly, the group was able to pick up valuable soft skills such as time management and communication in order to make this project possible.